Arquitectura de Computadores
Memoria: Trabajo final

Curso Arquitectura de Computadores
Grado en ingeniería Informática
Curso 2023-2024

Grupo 83
Equipo:6

Sandra Cea Rufián (100451130)
Daniel Fernandez Ocaña (100451063)
Marcos Caballero Cortes (100451047)
Lianru Jin Jiang (100451120)

Índice:

1. Diseño original.

Para el diseño original contamos con una primera aproximación de la propia estructura que se nos pide tener en nuestro trabajo, a continuación vamos a explicar cada una de estos componentes:

- **sim:** encontramos todos los componentes usados en programa principal, es decir:
  - **progargs.hpp**: La cabecera proporciona la interfaz para manejar los argumentos del programa. Encontramos una primera estructura de configuración que se usará a lo largo de la práctica con los argumentos formateados. También encontramos el propio constructor que vamos a utilizar durante el desarrollo de todo el proyecto y sus respectivos métodos para poder acceder a los argumentos
  - **progargs.cpp:** Encontramos la implementación de las funciones y los métodos de las clases que han sido declaradas en la parte de la cabecera. Entre ellos el manejo de los errores de no poder abrirlo por lectura, escritura,o argumentos incorrectos.
  - **grid.hpp:** En esta cabecera declaramos toda la parte propia de la partícula y resto de funciones necesarias para hacer cada uno de los puntos que se nos piden en el proyecto, es decir, la estructura de la partícula, la estructura que va a llevar la malla, las funciones necesarias para la densidad, aceleración, colisiones e interacciones con los límites del recinto. Además usamos templates que nos proporciona operaciones para la escritura y lectura binaria.
  - **grid.cpp:** Encontramos la implementación de todas estas funciones declaradas en la cabecera. Entre ellas la lectura del propio cuerpo de una partícula,la lectura del propio archivo,todo lo relacionado con la malla y las diferentes funciones como densidad, colisiones o aceleración entre otras que hay que tener en cuenta.
  - **block.hpp:** En esta cabecera encontramos la declaración de lo que es un bloque, es decir, la representación de las tres coordenadas, además de las variables para representarlas, y un vector llamado particles para almacenar identificadores de partículas. Además, hay un constructor que permite inicializar las coordenadas al crear un objeto Block.
  - **block.cpp:** Este archivo no los usamos dentro de nuestro diseño, ya que simplemente con declararlo en la cabecera, lo hemos implementado para usarlo dentro de grid.
  - **utils.hpp:**Por el contrario hemos añadido un nuevo archivo de cabecera en el que nosotros hemos creado una clase Vector3D en la que recopilamos todos las operaciones realizadas con vectores,además de la declaración de las constantes que se nos proporcionan en el documento.

En el fragmento de código que se enseña a continuación, podemos ver cómo para dichos vectores hemos usado una excepción. Esta excepción, se comentó previamente con el profesorado ya que no usamos memoria dinámica y seguía dando advertencia de clang-tidy y lo intentamos también con el *constexpr* pero seguía dando, por lo que decidimos dejarlo como lo teníamos y ponerle dicho comentario para que no saltara el error.

```
// NOLINTNEXTLINE(cert-err58-cpp)
const Vector3D external_acceleration(0.0, -9.8, 0.0); // Aceleracion externa
// NOLINTNEXTLINE(cert-err58-cpp)
const Vector3D bmax(0.065, 0.1, 0.065); // Limite superior de recinto
// NOLINTNEXTLINE(cert-err58-cpp)
const Vector3D bmin(-0.065, -0.08, -0.065); // Limite inferior de recinto
```

,

- **fluid:** Como se indica en la práctica, el fluid solo contiene la ejecución básica del programa, es decir inicializar a las clases necesarias y realizar las llamadas a las funciones.

- **utest:** En la parte de las pruebas unitarias está dividido en tres archivos (uno para cada parte del programa) para las diferentes pruebas de cada archivo:
    - **progargs_test.cpp:** En estas pruebas unitarias estamos verificando que los argumentos que se le pasan son correctos, el manejo de los errores de abrir para lectura o escritura entre otros, más adelante se comentarán de forma detallada.
    - **grid_test.cpp**: En estas pruebas unitarias estamos verificando si el número de partículas es el correcto o si por el contrario no concuerda con el que tiene que salir.
    - **utils_test.cpp:** Por el contrario, hemos añadido un nuevo archivo para pruebas unitarias de su respectivo archivo, que tiene que ver con las operaciones con vectores.
- **ftest:** En este componente encontramos las pruebas funcionales realizadas para la aplicación. Más adelante se explicará detalladamente la estructura del script utilizado.

En este apartado vamos a comentar también las decisiones que hemos ido tomando en el código y el por qué de estas.

- En cuanto a los reinterpret_cast, hemos usado la excepción permitida para que con el clang-tidy no diera error, en general, este error salta cuando hay conversiones que podrían potencialmente peligrosas.

```
template <typename T>
  requires(std::is_integral_v<T> or std::is_floating_point_v<T>)
char * as_writable_buffer(T & value) {
  // NOLINTNEXTLINE(cppcoreguidelines-pro-type-reinterpret-cast)
  return reinterpret_cast<char *>(&value);
}
```

Uso de herramientas basadas en IA

En cuanto a la parte del uso de herramientas basada en IA para realizar este proyecto hemos hecho uso de la herramienta Github Copilot mediante la licencia gratuita que nos ofrecen por ser estudiantes. El uso de esta herramienta nos ha sido muy útil para acomodarnos al uso del lenguaje de programación C++, lenguaje con el cuál teníamos una experiencia muy limitada. Gracias a haber utilizado Copilot, hemos podido descubrir algunas estructuras que no conocíamos de este lenguaje e investigar sobre su funcionamiento. Por ejemplo en situaciones en las que nos enfrentamos a conceptos poco familiares en C++, la herramienta genera fragmentos de código que no solo cumplían con nuestras necesidades, sino que también proporcionaban una oportunidad de aprendizaje al analizar y comprender cómo funcionaban estos fragmentos.

Otro caso de uso muy común, era tras realizar una parte del código que tenía mucha similitud a otra, como en el caso de las colisiones, las interacciones o incluso los tests, esta herramienta era capaz de entender que estábamos realizando con un par de líneas o simplemente con parte de una, y nos ayudó a completar algunas funcionalidades del código de manera más rápida, esto no solo aceleró el proceso de codificación, sino que también contribuyó a mantener una coherencia en la implementación.

En conclusión el uso de esta herramienta nos ha sido muy útil tanto para superar rápidamente la curva de aprendizaje asociada al lenguaje de programación C++, como para realizar tareas repetitivas de manera más productiva. En última instancia, la combinación de la inteligencia artificial y la supervisión humana ha enriquecido la experiencia de desarrollo, permitiéndonos abordar desafíos complejos de una manera más efectiva.

2. Optimización.

En este apartado vamos a comentar las decisiones que hemos ido tomando a lo largo del proyecto para poder optimizar el código y que su rendimiento y uso de energía mejorarán.

En un principio, diseñamos el código sin el uso de bloques ya que creíamos que habrían menos instrucciones y eso ayudaría al rendimiento de la caché ya que, en ese momento, parecían las decisiones más lógicas y adecuadas para el desarrollo del proyecto. Sin embargo, tras un análisis más profundo, mediante el uso de las herramientas “perf” y “valgrind”, descubrimos que causaba un problema crucial con el rendimiento de la caché.

El diseño sin bloques estaba generando un alto número de errores de caché, lo cual se traduce en accesos ineficientes a la memoria, y, por ende, en un rendimiento no óptimo del programa. Esto nos llevó a tener que rediseñar el código utilizando bloques para así poder mejorar significativamente la eficiencia del programa. Al utilizar bloques pasamos de tener un 15.13% de fallos de bifurcación(branch-misses) a tener un 0.22%.

En esta primera imagen, realizábamos el código sin bloques, por lo que se puede observar un alto porcentaje de fallos de bifurcación y un número bajo de instrucciones comparado con el código utilizando bloques.

```
392.502.991.828      instructions              #    0,99  insn per cycle
 56.588.660.153      branches                  #  484,739 M/sec
  8.561.834.444      branch-misses             #   15,13% of all branches
```

En esta segunda imagen reestructuramos el código utilizando bloques y se observa un número elevado de instrucciones y un porcentaje muy bajo de fallos de bifurcación.

```
352.829.402.890      instructions              #    2,58  insn per cycle
 71.330.520.339      branches                  # 1722,381 M/sec
    154.666.126      branch-misses             #    0,22% of all branches
```

- Una de las principales decisiones que hemos tomado a la hora de optimizar el código es el poder reducir lo máximo posible el número de bucles. En una primera instancia, hicimos un diseño del código que buscaba optimizar el uso de la caché lo máximo posible, por lo tanto hacíamos uso de estructuras e intentamos tener cuantos menos bucles anidados como pudimos, pero al ser la primera aproximación, no conseguimos el objetivo, de ahí que nos tardará más de lo normal, por lo que decidimos modificar esas partes del código donde existía este problema.

- Otra optimización que hemos llevado a cabo es precomputar las operaciones repetitivas con valores constantes, ya que tras analizarlo con Valgrind-cache observamos que consumían un 15% de la ejecución de nuestro programa, a continuación mostramos un ejemplo de ello:

```
Grid::Grid(struct File & file)
  : file(file),                                                    // file
    mass(fluid_density / pow(file.particles_per_meter, 3)),        // particle mass
    smoothing_lenght(radius_multiplicator / file.particles_per_meter),  // smoothing length
    smoothing_6(pow(smoothing_lenght, six)),  // smoothing length to the power of 6
    smoothing_6_pi_15(fifteen / (pow(smoothing_lenght, six) * M_PI)),
    smoothing_6_pi(pow(smoothing_lenght, six) * M_PI),
    smoothing_9(pow(smoothing_lenght, nine)), smoothing_2(pow(smoothing_lenght, 2)),
    pressure_rigdity_3(3 * pressure_rigidity),
    viscosity_45(forty_five * viscosity), fluid_density_2(2 * fluid_density), time_2(time_increase / 2),
    time_squared(pow(time_increase, 2)),
    density_transformation_constant((three_fifteen * mass) / (sixty_four * M_PI * smoothing_9)),
    viscosity_mass_h6_pi((viscosity_45 * mass) / (smoothing_6_pi)),
    mass_pressure((mass * pressure_rigdity_3) / 2) {
```

3. Pruebas realizadas:

grid_test.cpp

- Realizamos una serie de tests unitarios para la simulación del sistema, los tests son han sido escritos utilizando los la librería de google tests. A continuación, mostramos los test que se han llevado a cabo:
    - **GridTest** Es una clase que realiza funciones de configuración y limpieza con las funciones SetUp y TearDown. SetUp es el método que se ejecuta antes de cada test, que lo que hace es crear un fichero llamado “input.txt” con valores específicos para los tests, y TearDown es el método que se ejecuta después de cada test, que lo que hace es eliminar ese fichero “input.txt”.
    - **ReadFileTest:** En este test nos aseguramos de que la función readFile funciona correctamente y lee los archivos creados en la parte de SetUp realizando comparaciones utilizando ‘ASSERT_EQ’ para verificar que los datos leídos coincidan con los datos escritos
    - **GridConstructorTest:** Verificamos que el constructor de la clase Grid inicializa correctamente utilizando los datos leídos del archivo, calculamos los valores esperados para la masa de la partícula, la longitud de suavizado, el número de bloques y el tamaño de bloque, comparando estos con los valores en Grid.
    - **BlockIndexTest:** Este test verifica que el método blockIndex devuelve el índice del bloque correcto para una partícula dada, realizamos esto creando un objeto Grid e invocando el método blockIndex y luego comparamos los índices que hemos obtenido con los índices definidos en la clase Grid.
    - **InvalidNp y DifferentNp:** Estos dos tests verifican que la función readFile genera los mensajes de error correctos cuando tenemos un número de partículas incorrecto en el archivo. InvalidNp comprueba que readFile falla cuando el número de partículas es 0, y DifferentNp comprueba que readFile falla cuando el número de partículas encontradas no coincide con el número de partículas que hay en la cabecera.
    - **EvalDensitiesTest, EvalAccelerationsTest, CollisionsBlockTest, UpdateParticleTest e InteracitionsTest:** Estos test verifican que el estado entre las distintas etapas de nuestro programa es correcto, comparándola con las trazas correspondientes proporcionadas por los profesores. Para ello, tenemos primero el EvalDensitiesTest, el cual recibe el archivo “repos-base-1.trz” y el “densinc-base-1.trz". Con el primero,generamos un bucle en el que cada partícula pasa por nuestro programa y se actualiza, después con el segundo, comparamos los resultados de forma que si una partícula no tiene el mismo valor que en la siguiente traza, el test falla. En el resto de estos test, tenemos la misma funcionalidad, pero comparando las siguientes trazas con las respectivas operaciones.

progars test.cpp

- **ProgramArgumentsTest**: Como podemos ver en este archivo, podemos encontrar una clase que realiza funciones de configuración y limpieza con las funciones SetUp y TearDown.

- **ParseArgumentsTest:** Verifica que el método parseArguments de la clase ProgramArguments lee y analiza correctamente los argumentos de la línea de comandos. Además compruebas si los valores son los esperados.

- **CheckNStepsTest:** En este test se prueban con diferentes valores de entrada.Se comprueba si la función identifica correctamente casos válidos e inválidos y devuelve los códigos de error esperados.

- **ArgumentsTest:** En este test se prueban diversos escenarios para asegurarnos la validez de los argumentos de la línea de comando, se verifica el número correcto de argumentos, pasos de tiempo válidos y la existencia de archivos. Además, este test asegura que el programa finaliza con mensajes de error adecuados para casos inválidos.

- Y por último encontramos la función principal donde se inicializa Google Test y ejecuta todas las funciones definidas.

utils_test.cpp

- Estos test se han realizado para comprobar que el archivo que nosotros hemos diseñado para poder manejar el uso de operaciones aritméticas con vectores es correcta.A continuación explicamos los test que hemos realizado:
    - **Addition:**Verifica que la operación de suma (+) entre dos vectores devuelve el resultado esperado.
    - **Subtraction:**Verifica que la operación de resta (-) entre dos vectores devuelve el resultado esperado.
    - **ScalarMultiplication:**Verifica que la operación de multiplicación (*) entre dos vectores devuelve el resultado esperado.
    - **ScalarDivision:**Verifica que la operación de división (/) entre dos vectores devuelve el resultado esperado.
    - **DotProduct:**Verifica que la operación producto punto (dot) entre dos vectores devuelve el resultado esperado.
    - **Normalization:** Comprueba que se realiza correctamente la normalización de los vectores.
    - **Assigment:**Comprueba que la operación de asignación(=) de un vector a otro se realiza correctamente.
    - **UnaryMinus:**Comprueba que para los números negativos se realizan correctamente las operaciones entre dos vectores.
    - **CompoundAddition:** Verifica que la adición compuesta (+=) de un vector a otro se realiza correctamente.
    - **CompoundSubtraction:** Asegura que la resta compuesta (-=) de un vector a otro produce el resultado correcto.
    - **CompoundMultiplication:**Comprueba que la multiplicación compuesta (*=) de un vector por un escalar se realiza correctamente.
    - **CompoundDivision:**Valida que la división compuesta (/=) de un vector por un escalar devuelve el resultado esperado.
    - **Equality:**Verifica que la operación de igualdad (==) entre dos vectores devuelve el resultado esperado.
    - **Inequality:**Verifica que la operación de desigualdad (!=) entre dos vectores devuelve el resultado esperado.

ftest

- Para la parte de las pruebas funcionales de la aplicación hemos creado un script para realizar las pruebas de forma automatizada con diferentes conjuntos de argumentos. A continuación vamos a explicar que se comprueba en las diferentes partes del script:
    - En el primer test, se comprueba que el programa funciona cuando se le llama con el número correcto de argumentos. Para ello, comprobamos que con 0,1,2 y 4 argumentos, el output es el de un error.

- En el segundo test, comprobamos que si llamamos al programa con un número de timesteps no numérico, es decir, en nuestro caso con “test” en vez de un número, el programa imprime un código de error.
- En el tercer test, comprobamos que si llamamos al programa con el número de timesteps en negativo, el programa lanza un código de error.
- En el cuarto test, comprobamos que si llamamos al programa con un archivo de entrada que no existe, el programa lanza un código de error.
- En el quinto test, comprobamos que si llamamos al programa con un archivo de output que ya existe, el programa lanza un mensaje de error.
- Del test 6 al test 10, comprobamos que el programa funciona correctamente con 1, 2, 3, 4 y 5 interacciones con el archivo small.fld. Para ello, comprobamos el output de nuestro programa con el output esperado.
- Del test 11 al test 15, comprobamos que el programa funciona correctamente con 1, 2, 3, 4 y 5 interacciones con el archivo large.fld. Para ello, comprobamos el output de nuestro programa con el output esperado.

```
---------------
Testing FLUIDAPP
---------------


Test 1: Incorrect number of arguments
---------------------------------------------
fluidapp -> OK
fluidapp 4 -> OK
fluidapp 4 init.fld -> OK
fluidapp 4 init.fld final.fld 45 -> OK

Test 2: Time steps not numeric
---------------------------------------------
fluidapp test init.fld final.fld -> OK

Test 3: Negative time steps
---------------------------------------------
fluidapp -4 init.fld final.fld -> OK

Test 4: Non-existent input file
---------------------------------------------
fluidapp 4 non-existent.fld final.fld -> OK

Test 5: Existent output file
---------------------------------------------
fluidapp 4 small.fld -> OK

Test 6: Execution with 1 time steps
---------------------------------------------
fluidapp 1 small.fld -> OK

Test 7: Execution with 2 time steps
---------------------------------------------
fluidapp 2 small.fld -> OK

Test 8: Execution with 3 time steps
---------------------------------------------
fluidapp 3 small.fld -> OK
```

```
Test 9: Execution with 4 time steps
---------------------------------------------
fluidapp 4 small.fld -> OK

Test 10: Execution with 5 time steps
---------------------------------------------
fluidapp 5 small.fld -> OK

Test 11: Execution with 1 time steps
---------------------------------------------
fluidapp 1 large.fld -> OK

Test 12: Execution with 2 time steps
---------------------------------------------
fluidapp 2 large.fld -> OK

Test 13: Execution with 3 time steps
---------------------------------------------
fluidapp 3 large.fld -> OK

Test 14: Execution with 4 time steps
---------------------------------------------
fluidapp 4 large.fld -> OK

Test 15: Execution with 5 time steps
---------------------------------------------
fluidapp 5 large.fld -> OK
```

4. Evaluación de rendimiento y energía:

Optimización de caché

En cuanto a las optimizaciones de caché, observamos que podríamos mejorar el rendimiento al alterar la forma de recorrer los bloques y sus respectivas partículas. En un inicio recorríamos todos los bloques, y para cada bloque todas sus partículas, y evaluábamos si el índice de la partícula es menor que la del bloque contiguo, aplicamos la función de densidad o de aceleración.

Esto provocaba que tuviéramos que recorrer todos los bloques antes de aplicar la transformación a las densidades, y por lo tanto teníamos que separar esto en dos bucles. Lo mismo pasaba con la aceleración y sus respectivas colisiones entre partículas y con el eje y su reposicionamiento. Tras actualizar al nuevo modelo de recorrer bucles donde juzgamos si el índice de un bloque es menor que el otro, reducimos el número de accesos, instrucciones y nos permitió anidar dos bucles, lo que se traduce en un mejor rendimiento debido a la reducción de fallos de caché:

Versión inicial	Versión final
	

```
void Grid::evalDensities() {
  for (Block & block_i : blocks) {
    Vector3D position               = Vector3D(block_i.x, block_i.y, block_i.z);
    std::vector<Vector3D> neighbours = getNeighbours(position);
    for (size_t particle_i : block_i.particles) {
      for (Vector3D neighbour : neighbours) {
        auto block_j = static_cast<size_t>(neighbour.x + neighbour.y * n_blocks_x +
                                           neighbour.z * n_blocks_x * n_blocks_y);
        for (size_t particle_j : blocks[block_j].particles) {
          if (particle_i < particle_j) { evaluateDensities(particle_i, particle_j); }
        }
      }
    }
  }

  for (struct Particle & particle : file.particles) {
    particle.density = (particle.density + smoothing_6) * density_transformation_constant;
  }
}

void Grid::evalAccelerations() {
  for (Block & block_i : blocks) {
    Vector3D position               = Vector3D(block_i.x, block_i.y, block_i.z);
    std::vector<Vector3D> neighbours = getNeighbours(position);
    for (size_t particle_i : block_i.particles) {
      for (Vector3D neighbour : neighbours) {
        auto block_j = static_cast<size_t>(neighbour.x + neighbour.y * n_blocks_x +
                                           neighbour.z * n_blocks_x * n_blocks_y);
        for (size_t particle_j : blocks[block_j].particles) {
          if (particle_i < particle_j) { evaluateAccelerations(particle_i, particle_j); }
        }
```

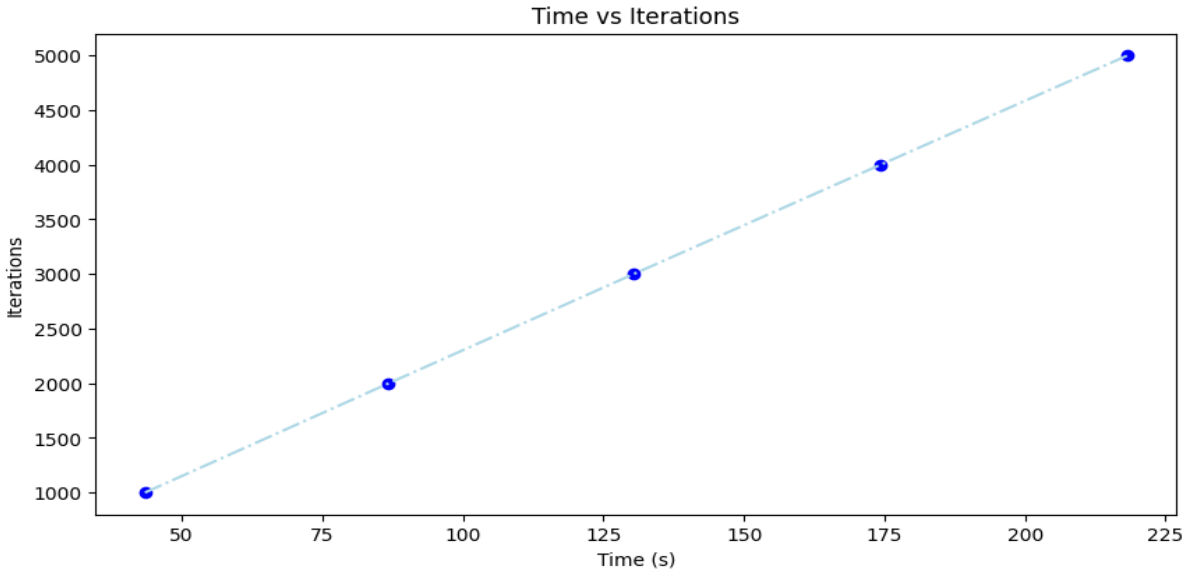
```
void Grid::makeSimulation() {
  for (size_t i = 0; i < file.particles.size(); i++) {
    file.particles[i].density      = 0;
    file.particles[i].acceleration = external_acceleration;
    Vector3D const index                = blockIndex(file.particles[i]);
    auto position =
        static_cast<size_t>(index.x + index.y * n_blocks_x + index.z * n_blocks_x * n_blocks_y);
    blocks[position].particles.push_back(i);
  }

  for (size_t block_i = 0; block_i < blocks.size(); block_i++) {
    evalDensities(block_i);
    transformDensities(blocks[block_i]);
  }

  for (size_t block_i = 0; block_i < blocks.size(); block_i++) {
    evalAccelerations(block_i);
    collisions(blocks[block_i]);
    updateParticle(blocks[block_i]);
    interactions(blocks[block_i]);
    blocks[block_i].particles.clear();
  }
}
```

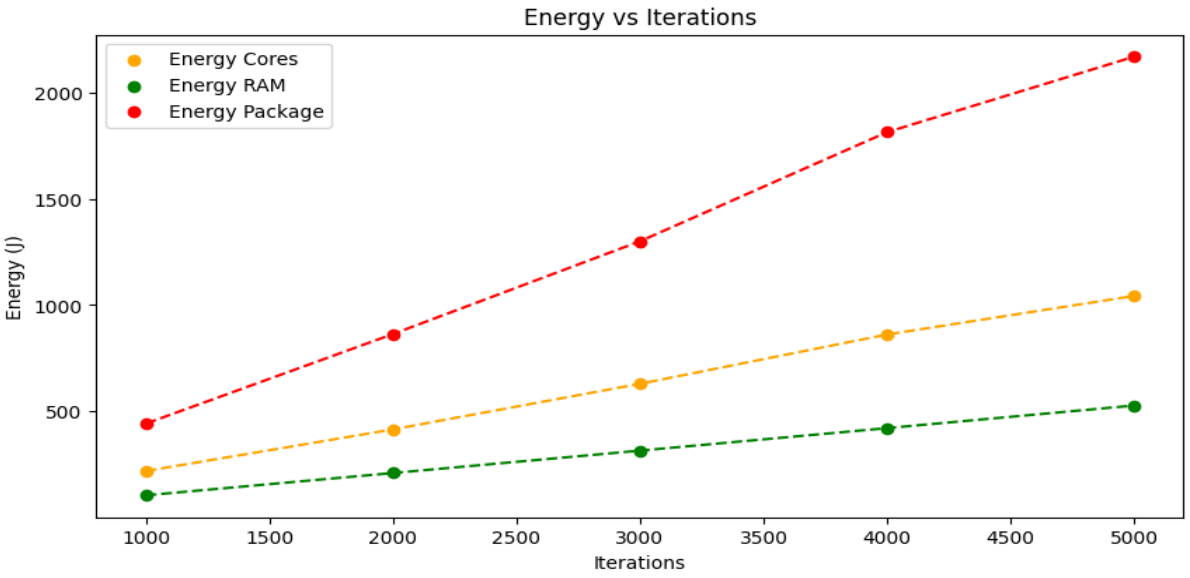
Optimización de rendimiento y energía

Una vez acabamos el código, evaluamos la memoria caché utilizando cachegrind y cg_annotate, donde nos dimos cuenta de que la función std::abs que utilizabamos en la versión sin bloques ocupaba un 11% del total del programa, a pesar de ser esta solo una línea, tras eliminar su uso comprobamos que la mayor parte de las instrucciones de nuestro programa consistían en la función std::pow, y la mayoría de estas operaciones las podríamos precomputar como se indicó previamente. Tras realizar estos pasos conseguimos mejorar el tiempo de ejecución de 53s a 41s.

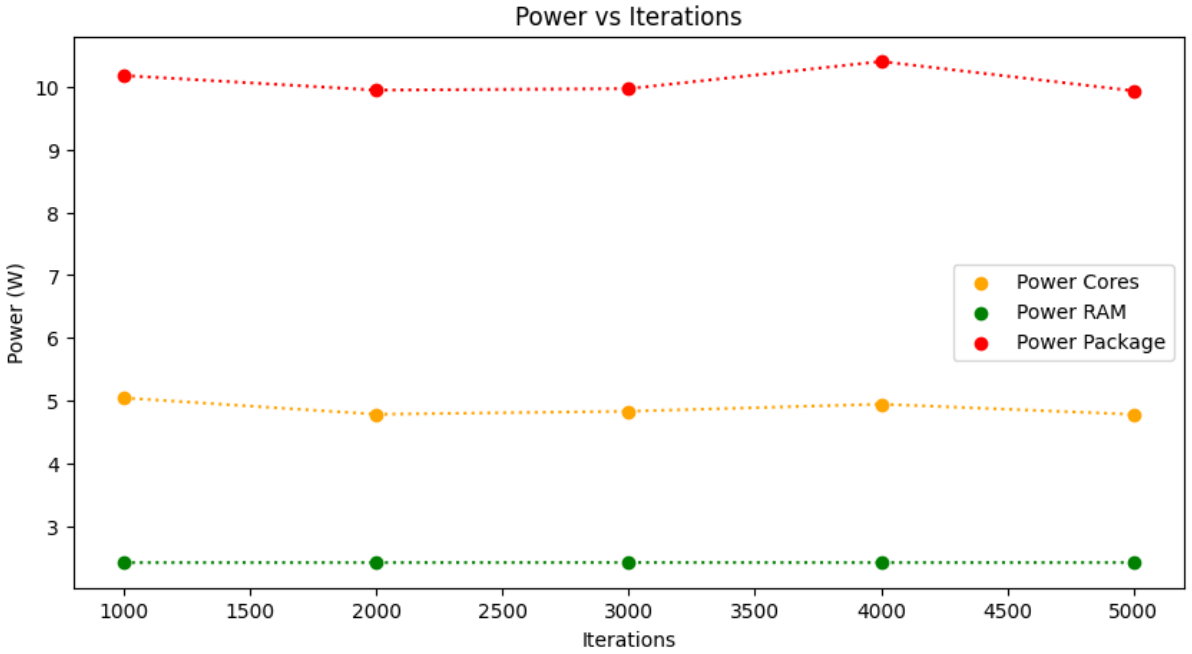
```
Performance counter stats for './build/fluidapp 1000 ./inputs/large.fld ./outputs/large.fld':

         41.413,90 msec task-clock                #    0,998 CPUs utilized
               181      context-switches          #    0,004 K/sec
                 0      cpu-migrations            #    0,000 K/sec
             1.436      page-faults               #    0,035 K/sec
   137.017.461.477      cycles                    #    3,308 GHz
   352.829.402.890      instructions              #    2,58  insn per cycle
    71.330.520.339      branches                  # 1722,381 M/sec
       154.666.126      branch-misses             #    0,22% of all branches

      41,482724154 seconds time elapsed

      41,414919000 seconds user
       0,000000000 seconds sys
```

Utilizando el comando *perf stat -e power/energy-cores/, power/energy-ram/, power/energy-gpu/, power/energy-pkg/ ./build/fluidapp {i}000 ./inputs/large.fld ./outputs/large-{i}000.fld* siendo "i" números del 1 al 5 dependiendo del script generamos los resultados de realizar i*1000 iteraciones para evaluar su rendimiento y consumo de energía y obtuvimos los siguientes resultados.





A partir de la energía y el tiempo calculamos la potencia



Como se puede observar en las imágenes anteriores, el tiempo crece de manera lineal con el número de iteraciones, lo que indica que cuantas más iteraciones más tiempo va a consumir en todos los casos.

Por otro lado, podemos observar que tanto para la energía consumida por la CPU como la RAM, crecen de manera lineal y uniforme, lo que indica que están contribuyendo ambas al consumo de energía general.

También podemos observar que la CPU hace más uso de la energía que la RAM, lo que indica que la mayor carga de trabajo del programa se encuentra en el uso de instrucciones y no en los accesos a memoria.

Respecto a la potencia podemos llegar a la misma conclusión que con la energía, es decir, la mayor carga de trabajo de nuestro programa se encuentra en las instrucciones y no en los accesos a memoria.

5. Organización del trabajo:

Breve descripción de la tarea	Persona encargada	Tiempo estimado
Pensar estructura inicial y actualizar los archivos del trabajo	Sandra	1 hora y 30 minutos
Creación progrars.cpp y progrars.hpp	Daniel	2 horas y 30 minutos
Creación de tests para progrars	Marcos	1 hora y 30 minutos
creación main fluid	Sandra	30 minutos
CMakelists que funciona	Daniel	1 hora y 10 minutos
Reestructuración de progrargs	Lianru	40 minutos
Añadir test progrargs que funcionan	Daniel	1 hora y 20 minutos
bug, cambiamos en progrargs std::cout por std::cerr y arreglar tests	Lianru	1 hora
cambios en fluid.cpp creación contador de particulas	Marcos	1 hora y 30 minutos
implementación para leer el archivo en grid.cpp, y en grid.hpp	Daniel	1 hora 30 minutos
solucionado problema de Readfile que no funcionaba con sus tests y añadir la librería lib	Sandra	1 hora
Definidos e incializados los parámetros de grid y añadidos tests para grid	Daniel	3 horas
Añadir restricciones para la lectura de partículas en grid y creación de más tests	Lianru	2 horas
solución de errores en los tests de grid, funcionan todos	Sandra	1 hora
Creación de la iniciación de simulación	Lianru	2 horas
Generar test para grid, progargs y utils	Marcos	2 horas y 30 minutos

Limpiar errores de Clang-tidy en progargs, fluid y grid	Daniel	2 horas
Limpiar los errores de Clang-tidy en tests	Sandra	30 minutos
Terminar la implementación de la funicionalidad	Daniel	3 horas
Añadir tests para comprobar los archivos grandes	Marcos	1 hora y 30 minutos
Arreglar la salida del programa	Lianru	30 minutos
Cambiar block.hpp	Daniel	1 hora
Refactorizar grid	Daniel	2 horas
Crear scripts run.sh y build.sh para su uso en avignon	Daniel	30 minutos
Buscar error en código y arreglar la inicialización de aceleración y densidad	Marcos	40 min
Realizar cambios en las operaciones más usadas para la optimización de la caché	Lianru	2 horas
Realizar cambios en la funcionalidad del código para no comprobar cosas innecesarias y realizar cambios necesarios para la optimización de caché	Daniel	4 horas
Optimización de bucles en grid.cpp	Lianru	2 horas
Explorar y hacer cambios en cmake mediante las flags y en bucles.	Marcos	2 horas
Terminar de refactorizar el código	Daniel	30 minutos
Terminar de formatear con clang format	Sandra	10 minutos
Arreglar los test funcionales que fallaban	Marcos	2 horas
Crear test para comprobar el resultado con las trazas	Daniel	2 hora y 30 minutos
Realizar pruebas de rendimiento y energía en	Sandra	45 minutos

avignon		

6. Conclusiones.

Dani: Considero que este proyecto presenta un conjunto de desafíos interesantes al situarnos en la posición de trabajar con un cliente cuyo análisis de requisitos ha sido, en muchos casos, propenso a errores y ambigüedades. La dinámica del proyecto ha experimentado cambios frecuentes a lo largo del tiempo, lo cual ha requerido una adaptación continua. Estos aspectos, aunque han planteado dificultades, han contribuido a mi desarrollo profesional al permitirme enfrentar situaciones que podrían surgir en un entorno laboral. Además, he tenido la oportunidad de fortalecer mis habilidades de comprensión en un contexto práctico. Por otro lado cabe agradecer la gran labor del profesorado y principalmente del coordinador durante esta práctica; tanto el foro, como las tutorías han sido de gran ayuda, y además tanto las trazas como los resultados esperados han sido de vital importancia a la hora de enfrentar un problema durante el desarrollo.

Sandra: Comprendo que la dinámica de modificaciones frecuentes en el documento base (PDF) ha suscitado cierto descontento. No obstante, es reconfortante observar que el equipo ha abordado esta situación con resiliencia, convirtiéndola en una oportunidad para desarrollar habilidades de adaptabilidad y gestión de cambios.

Lianru: En general este proyecto ha proporcionado una valiosa experiencia en el desarrollo de una simulación de fluidos mediante el método de hidrodinámica de partículas suavizadas. A lo largo de este proyecto ha habido varios problemas por los constantes cambios en el enunciado de la práctica, pero esto también es una manera de adaptarse a cambios no previstos en la vida laboral. Este trabajo, además de haber fortalecido mis habilidades en programación eficiente me ha ayudado a tener una nueva perspectiva sobre la importancia de la optimización.

Marcos: Este proyecto me ha proporcionado un contexto bastante amplio sobre cómo afrontar una situación laboral en la que se van cambiando cada poco tiempo las especificaciones, haciendo que tuviésemos que cambiar alguna parte de nuestro código o planteamiento. Respecto a la práctica en general, me ha servido para terminar de comprender como usar correctamente C++ y Linux, las cuales son dos tecnologías que usaré en un futuro cercano en el ambiente laboral.

Conclusiones generales

En una primera instancia, cabe destacar que el trabajo ha resultado difícil de entender, partiendo de la premisa que el documento al igual que el contenido que se nos proporcionaba se modificaba cada poco tiempo sin especificar en qué zonas se cambiaba con respecto a la versión anterior.

Como consecuencia de esto, se modificó la entrega de la práctica en un primer momento y es algo con lo que estamos muy agradecidos, aun así respecto a lo que se imparte en clase de lenguaje de C++ y el nivel que se pide en la práctica es muy diferente, ya que para el trabajo era necesario tener cierto nivel de comprensión sobre lo que se nos está pidiendo exactamente y cierto nivel a la hora de programar en un lenguaje como este.

También queremos comentar el problema que ha habido con CLion. Como hemos podido comprobar en muchos de los grupos que han realizado la práctica al igual que nosotros, CLion no funcionaba a todos los miembros del grupo por igual, ya que había que configurarla en cada uno de los dispositivos de cada uno de los miembros del grupo y se perdía mucho tiempo, por lo que para no perder tiempo, hicimos todo desde Visual Studio Code, añadiendo las versiones adecuadas al clang-tidy y a los cmake para que todo fuera igual que si estuviéramos trabajando en CLion.

En cuanto al rendimiento cabe destacar que hemos tenido en cuenta la diferencia de hacer la práctica usando bloques y sin usar bloques. Evidentemente y después de haberlo comprobado, es más eficiente el uso de bloques, ya que el número de instrucciones ha disminuido en comparación con cuando no utilizamos bloques, y el número de fallos de bifurcación es mucho menor por lo que hace el código más óptimo. Cabe destacar el buen uso de optimización de código que hemos hecho, consiguiendo que el tiempo de rendimiento haya bajado considerablemente desde que empezamos a ejecutarlo.

Respecto a todos estos acontecimientos que se han ido mostrando a lo largo del curso en sus respectivos tiempos, nuestra opinión acerca de la práctica, es que resulta fácil de entender una vez se tiene claro que es lo que se está pidiendo, y en general faltaba esa parte para poder hacer el trabajo de una forma clara, concisa y sencilla.

Para los próximos años, esperamos que se tengan en cuenta todos los inconvenientes que han sucedido y que sea más sencillo el realizar una práctica de este estilo.